Chapter 1

The Quality Strategy for UML

Quality—you know what it is, yet you don't know what it is. But that is self contradictory. . . . But some things are better than others, that is, they have more quality. . . . But if you can't say what Quality is, how do you know what it is, or how do you know that it even exists? If no one knows what it is, then for all practical purposes it doesn't exist at all. But for all practical purposes it really does exist. . . . So round and round you go, spinning mental wheels and nowhere finding any place to get traction. What the hell is Quality? What is it?[1]

CHAPTER SUMMARY

This chapter discusses the underlying concepts of modeling and the effect of verification and validation (V&V) techniques on its quality. After creating an understanding of modeling and its relevance to quality, this chapter describes the toolbox of Unified Modeling Language (UML) diagrams and divides them over the three distinct yet related modeling spaces: problem, solution and background. Following this is a discussion on defining the syntax, semantics and aesthetics checks for V&V of UML models and how their levels and skill sets affect the quality of the project.

[1]From Robert Pirsig's all-time favorite, *Zen and the Art of Motorcycle Maintenance* (http://www.nobigv.tripod.com). For an irresistibly different perspective, read *The Zen Manifesto* (www.osho.com).

1.1 MODELING AND QUALITY

1.1.1 The Modeling Advantage

Modeling enhances quality because it enhances communication. Through modeling, communication becomes efficient and effective. This is so because modeling raises abstraction to a level where only the core essentials matter. The resultant advantage is twofold: easier understanding of the reality that exists and efficient creation of a new reality (Unhelkar, 1999).

The advantage of modeling in understanding complexity is derived from the fact that models *distill* reality. Elements that are not significant in understanding the reality are dropped. Modeling also fosters creativity by focusing on the essentials and ignoring the gory details. This holds true for modeling in many industries such as construction, medicine and transportation. However, the role of modeling is even more important in software development, where it provides the means of understanding existing software systems whose legacy nature renders them extremely complex, as well as in developing and customizing new software systems expected to serve highly demanding customers in constant flux.

Consider, for example, an abstraction of a COBOL application. Modeling assists in understanding that application and the complex environment in which it operates. Creating a model, however brief, is imperative in understanding the traditional legacy application.

Modeling also facilitates smoother creation of the new reality. For example, creating a model of a software system is much easier, cheaper and faster than creating the actual system. Once the concepts are bedded down, they can be adorned with all the additional paraphernalia that makes up the final application. In this process, modeling not only represents what we want, but also educates us in understanding what we *should* want. It is not uncommon to have a user *change* her requirements based on a prototype (type of model) of a system that she has seen. Alterations and additions to the functionality required of the system during early modeling stages are welcome signs, providing significant impetus to the modeling and quality activities within a project. This is because changes during the early modeling stages of a software life cycle are cheaper and faster to incorporate or fix than those introduced later during implementation.

1.1.2 Modeling Caveats

Despite the stated and obvious advantages of modeling, there is one singularly important factor that influences the value of a model: *the quality of the model itself*. If the abstraction is incorrect, then obviously the reality eventually created out of that abstraction is likely to be incorrect. An incorrect abstraction will also not reflect or represent the reality truthfully. Therefore, model quality is of immense importance in eventually deriving quality benefits.

Modeling is limited by the following caveats:

- A model, by its very nature, is an abstraction of the reality. The modeler, depending on her needs, keeps parts of the reality that are important to her in a particular situation and leaves out others which may be considered less important. Therefore, the model is not a complete representation of the reality. This leads to the possibility of the model's being subjected to different interpretations.
- Unless a model is dynamic, it does not provide the correct sense of timing. Since the reality is changing, it is imperative that the model change accordingly. Otherwise, it will not be able to convey the right meaning to the user.
- A model may be created for a specific situation or to handle a particular problem. Needless to say, once the situation has changed, the model will no longer be relevant.
- A model is a singular representation of possible multiple elements in reality. For example, a class in software modeling parlance is a single representation of multiple objects. In such cases, a model may not provide a feel for the operational aspects of the system, such as volume and performance.
- The user of the model should be aware of the notations and language used to express the model. For example, when the design of a house is expressed using a paper model, it is necessary for the user to know what each of the symbols means. Nonstandard notations and processes can render a model useless.
- Modeling casually, or at random, without due care and consideration for the nature of the models themselves, usually results in confusion in projects and can reduce productivity. Therefore, formality in modeling is necessary for its success.
- Models must change with the changing reality. As applications and systems change, so should the models if they are to be relevant. Models that are not kept up-to-date can be misleading.
- Processes play a significant role in steering modeling activities. Modeling without considering processes is a potential practical hazard and should be avoided.

Goals, methods and performance are considered the three major aspects of quality by Perry (1991).

1.1.3 Context of Model Quality

Where and how should the model quality effort be focused? Firstly, we must understand that model quality is not the only aspect of quality in a project. Model quality exists within the context of other quality dimensions or levels, and these influence each other as well as model quality. In practical UML-based projects, the following levels of quality are listed by Unhelkar (2003):

- *Data quality*—the accuracy and reliability of the data, resulting in quality work ensuring integrity of the data.

- *Code quality*—the correctness of the programs and their underlying algorithms.
- *Model quality*—the correctness and completeness of the software models and their meanings.
- *Architecture quality*—the quality of the system in terms of its ability to be deployed in operation.
- *Process quality*—the activities, tasks, roles and deliverables employed in developing software.
- *Management quality*—planning, budgeting and monitoring, as well as the "soft" or human aspects of a project.
- *Quality environment*—all aspects of creating and maintaining the quality of a project, including all of the above aspects of quality.

1.1.4 Model Quality

The aforementioned quality levels play a significant role in enhancing the overall quality of the output of a software project. Most literature on quality, however, focuses on code and data quality. Even when modeling appears in the discussion of quality, it is with the aim of creating good-quality software (data and algorithms). In this book, however, model quality refers to the quality of the software models themselves. Model quality depends on detailed V&V of those models.

In software projects without substantial modeling, code remains the primary output of the developers. In such projects, code emerges from the developer's brain—directly. This, as the history of software development indicates (Glass, 2003), has had disastrous effect on software projects.

Quality-conscious software projects use modeling throughout the entire life cycle. Subsequently, modeling is used not only to create the software solution but also to understand the problem. As a result, modeling occurs in the problem, solution and background (architectural) spaces. The modeling output in such software projects transcends both data and code and results in a suite of visual models or diagrams. While these models go on to improve the quality of the code produced, it is not just their influence on the implemented code that interests us but also their own quality—that is, the quality of the models themselves. There is an acute need to subject the software models themselves to quality assurance and quality control processes. It is important that these models adhere to known standards and are also subjected to stringent quality control. Model quality is all about V&V of the models themselves. The result is not only improved model quality, but also improved communication among project team members and among projects.

1.2 POSITIONING UML FOR MODELING

How do we build software models? The ubiquitous flowcharts, followed by the entity relationship (E-R) and data flow diagrams (DFDs), are no longer sufficient to model modern software systems. With the advent of objects and components, Web services and grid computing, and pervasive mobile computing, we need a sophisticated as well as an exhaustive suite of modeling techniques. UML version 2.0 (Object Management Group [OMG]) is specifically advocated as a software *modeling* language for visualization, specification, construction and documentation (Booch et al., 1999). Thus, it is *not* a programming language, although with the recent initiatives involving model-driven architecture (MDA), we are close to using UML as an executable UML language. Thus, overall, UML, together with a programming language for implementation, provides an excellent mechanism to develop software systems.

Furthermore, it is worth noting that UML is not a methodology, but rather a common and standard set of notations and diagrams. These are used by processes in varying ways to create the required models in the problem, solution and background modeling spaces. Therefore, the checklists developed later in this book for V&V purposes are likely to vary, depending on the process; in other words, the V&V checklists will change, depending on the role and the modeling space in which they are applied.

It is also worth mentioning that UML has been (and is being) used effectively in a large range of projects, including:

- New development projects, where systems are designed from scratch and the new business applications are modeled using, for example, UML's use cases and activity diagrams.
- Integration projects, where newer systems—typically Web-enabled systems—are integrated with existing (typically legacy) systems.
- Package implementation, where UML's behavioral diagrams can be used to understand the requirements of the implementation of the customer relationship management system (CRMS) or the enterprise resource planning (ERP) system.
- Outsourcing projects, where UML provides the basis for scoping, delivery and testing.
- Data warehousing and conversion projects, where not only are the data and related information modeled using UML, but the conversion and testing processes also use UML to document the flow.
- Educational projects, where UML can be used for testing concepts, for example for teaching and learning object orientation.

In addition to the above types of projects, UML is being used in small, medium-sized and large projects (Unhelkar, 2003). Due to such wide-ranging applicability of UML, the model quality of UML-based projects assumes great importance. Let us, therefore, consider UML from a model quality perspective.

1.3 QUALITY ASPECTS OF UML

UML has four main purposes: visualization, specification, construction and documentation (Booch et al., 1999). Therefore, in investigating the quality of UML models, it is worthwhile to consider how these factors affect, and are affected by, quality.

Visualizing—UML notations and diagrams provide an excellent industry standard mechanism to represent pictorially the requirements, solution and architecture. UML's ability to show business processes and software elements visually, spanning the entire life cycle of software development, provides the basis for extensive modeling in software development. UML, through its class representations, can bring the reality (real customers, accounts and transactions in a typical banking system) close to the people working in the solution space by modeling the corresponding `Class Customer`, `Class Account` and `Class Transaction`. The small gap between models and reality, especially in object-oriented (OO) development, improves the quality of visualization. This quality of visualization is enhanced not only by the use of UML as a standard, but also because of the large number of Computer Aided Software Engineering (CASE) tools supporting these visual diagramming techniques. CASE tools in modeling facilitate the work of teams of modelers and prevent syntax errors at the visual modeling level.

Specifying—Together with visual representations, UML facilitates the specification of some of its artifacts. For example, specifications can be associated with the actors, use cases, classes, attributes, operations and so on. These UML specifications help enhance the quality of modeling, as they enable additional descriptions of the visual models, enable members of a project team to decide which areas of a particular diagram or element they want to specify, and allow them (through CASE tools) to make the specifications available to all stakeholders. The specifications can be made available in various formats, such as a company's intranet Web page, a set of Word documents or a report.

Constructing—UML can also be used for software construction, as it is possible to generate code from UML visual representations. This is becoming increasingly important with the rapidly advancing concepts of executable UML (Mellor and Balcer, 2002) and the MDA initiative (Mellor et al., 2004). A piece of software that is constructed based on formal UML-based modeling is likely to fare much better during its own V&V. Classes and class diagrams, together with their specifications (e.g., accessibility options, relationships, multiplicities), ensure that the code generated through these models is correctly produced and is inherently superior to hand-crafted code (i.e., code without models).

Documenting—With the help of UML, additional and detailed documentation can be provided to enhance the aforementioned specifications and visual representations. Documentation has become paramount—not only the type that accompanies the code, but also the type that goes with models, prototypes and other such artifacts. In UML, diagrams have corresponding documentation, which may be separate from the formal specifications and which goes a long way toward explaining the intricacies of visual models.

1.4 UNDERSTANDING MODELING SPACES IN SOFTWARE

With the aforementioned four dimensions in which UML promises quality enhancement, it is still vital to remember that UML-based modeling does not happen within a single modeling space. Successful modeling needs to consider the *areas* in which modeling needs to take place. These modeling spaces have been formally considered and discussed by Unhelkar and Henderson-Sellers (2004). This role-based division is shown in Figure 1.1.

This figure depicts the three distinct yet related modeling spaces: problem, solution and background. These role-based divisions form the basis of further quality V&V work with respect to UML models. These divisions provide a much more robust approach to quality modeling, as they segregate the models based on their purpose, primarily whether the model is created to understand the problem, to provide a solution to the problem, or to influence both of these purposes from the background, based on organizational constraints (e.g., stress, volume and bandwidth), and need to reuse components and services.

1.5 MODELING SPACES AND UML

The modeling spaces shown in Figure 1.1 can be specifically considered within the context of UML. To ensure the quality of UML-based models and to apply the correct V&V checklists to those models, it is essential to focus on the objectives of the modeling exercise and use the UML-based diagrams that will help the modeler achieve these objectives. Thus, the applicability of UML diagrams differs from

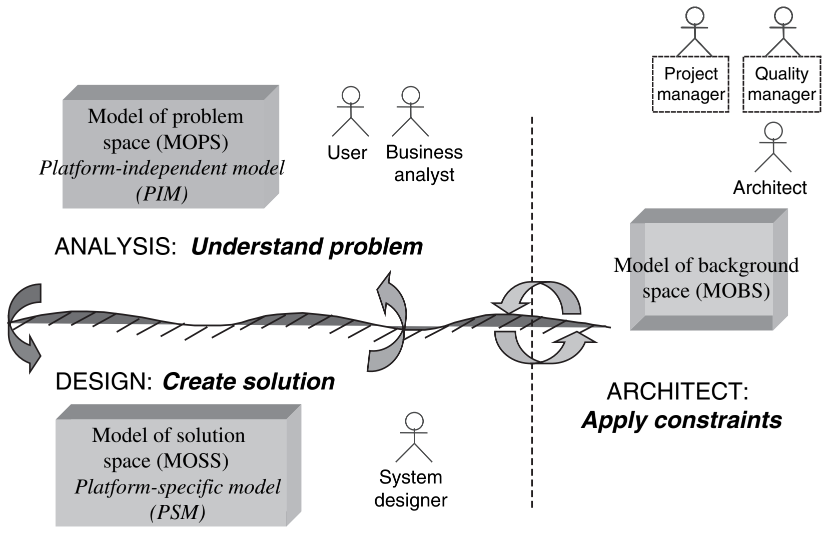
Project manager
Quality manager
Model of problem space (MOPS) Platform-independent model (PIM)
User
Business analyst
Architect
ANALYSIS: Understand problem
Model of background space (MOBS)
DESIGN: Create solution
ARCHITECT: Apply constraints
Model of solution space (MOSS) Platform-specific model (PSM)
System designer

Figure 1.1 Software modeling spaces and the modeling work of analysis, design, and architecture in them.

project to project. The intensity of the application of these diagrams in creating the models also differs, depending on the reasons for modeling when creating the diagrams. The purpose for creating the diagrams has a direct bearing on the way they are created, extended, and, of course, verified and validated. The modeling spaces are extremely helpful in clarifying the purpose of modeling, particularly its role.

1.5.1 Importance of UML Diagrams to Respective Models

The modeling spaces discussed in the previous subsection were the problem, solution and background spaces. One should expect to encounter these modeling spaces in some form in any modeling exercise. We consider them here specifically in UML-based modeling. For ease of usage and discussion, the three modeling spaces are as follows:

MOPS: model of problem space
MOSS: model of solution space
MOBS: model of background space

These models are shown in their corresponding modeling spaces in Figure 1.1. Also shown in the figure are the various primary roles in the modeling spaces—namely, the user, business analyst, system designer, system architect, project manager and quality manager. Of these roles, the three that work in the modeling spaces to create MOPS, MOSS, and MOBS are the business analyst, the designer and the architect. We now also consider the UML diagrams themselves in order to understand their appropriateness to the modeling spaces.

1.5.2 List of UML Diagrams

In order to "spread" UML diagrams in the appropriate modeling spaces, and with the eventual aim of applying V&V checks to them, we now consider the diagrams that make up UML. Table 1.1 lists the UML 2.0 diagrams (based on OMG and on Fowler, 2003).

While the list in Table 1.1 is not comprehensive, it is worth mentioning that the number of diagrams existing in the UML literature even in its earlier versions varies. For example, Booch et al. (1999) listed only nine diagrams. However, Jacobson et al.'s earlier work (1992), as well as that of Rosenberg and Scott (1999), (who list robustness diagrams separately), had a different number of diagrams from that on the OMG's list.

Increasingly, practitioners have started listing the package diagram separately from the class diagram because of its increasing importance in organizational and architectural areas of the system. The package diagram is accepted as a separate diagram in the current UML 2.0 literature. Sometimes the sequence and communication

TABLE 1.1 UML 2.0 Diagrams

UML Diagrams	Represent
1. Use case	functionality from the user's viewpoint
2. Activity	the flow within a Use case or the system
3. Class	classes, entities, business domain, database
4. Sequence	interactions between objects
5. Interaction overview	interactions at a general high level
6. Communication	interactions between objects
7. Object	objects and their links
8. State machine	the run-time life cycle of an object
9. Composite structure	component or object behavior at run-time
10. Component	executables, linkable libraries, etc.
11. Deployment	hardware nodes and processors
12. Package	subsystems, organizational units
13. Timing	time concept during object interactions

diagrams (as named in UML 2.0, these are the collaboration diagrams of the earlier UML versions) are listed together as interaction diagrams.

The component and deployment diagrams are also referred to as "implementation" diagrams in UML literature. Object diagrams are theoretically treated as independent diagrams in their own right but are often not supported by CASE tools—resulting in their being drawn underneath communication diagrams within CASE tools. While this discussion introduces you to the range of UML diagrams, it is more important to know the diagrams' precise strengths and the purpose for which they can be used rather than focus on the precise list.

1.5.3 UML Diagrams and Modeling Spaces

Table 1.1 summarizes the UML diagrams and the modeling aspect of software solutions represented by them. These diagrams have a set of underlying rules that specify how to create them. The rigor of these rules is encapsulated in what is known as the OMG's "meta-model". The meta-model also helps to provide rules for cross-diagram dependencies.

The importance of the meta-model is that it renders the UML *elastic*—it can be stretched or shrunk, depending on the needs of the project. This is discussed further in Chapter 2. Because of the elasticity of UML, the extent and depth to which the UML diagrams are applied in creating models are crucial to the success of projects using UML. Not all diagrams apply to all situations. Furthermore, not all diagrams are relevant to a particular role within a project. As Booch et al. (1999) correctly point out:

> Good diagrams make the system you are developing understandable and approachable. Choosing the right set of diagrams to model your system forces you to ask the right questions about your system and helps to illuminate the implications of your decisions.

Therefore, we must first select the right set of diagrams for a particular modeling space. This is achieved next, followed by a discussion on the right level of application of these models. A series of V&V criteria are then applied to these diagrams in subsequent chapters. While the UML diagrams apply to all modeling spaces, Table 1.2 next summarizes the relative importance of each of the UML diagrams to each of the modeling spaces and major modeling roles within the project. While project team members can work in any of these modeling spaces using any of the UML diagrams, good models are usually the result of understanding the importance of the diagrams with respect to the modeling spaces. This is shown in Table 1.2. As is obvious from this table, modelers are not prevented from using any of the UML diagrams. However, Table 1.2 provides *focus* in terms of using UML diagrams for a particular role within a project. This information can be invaluable in organizing the quality team, as well as in following the process that will verify and validate these diagrams.

The categorization in Table 1.2 ensures a balance between the desire to use everything provided by UML and the need to use only the relevant diagrams as a good starting point for a modeling exercise. Table 1.2 uses the "*" rating to indicate the importance and relevance of UML diagrams within MOPS, MOSS and MOBS. A maximum rating of ***** is provided for diagrams of the utmost importance to the model in a particular space.

1.5.4 Model of Problem Space (MOPS)

Figure 1.1 shows MOPS in the problem space. In UML projects, MOPS deals with creating an understanding of the problem, primarily the problem that the potential user of the system is facing. While usually it is the business problem that is being

TABLE 1.2 Importance of UML Diagrams to Respective Models (a Maximum of Five * for Utmost Importance to the Particular Space)

UML Diagrams	MOPS (Business Analyst)	MOSS (Designer)	MOBS (Architect)
Use case	*****	**	*
Activity	*****	**	*
Class	***	*****	**
Sequence	****	*****	*
Interaction overview	****	**	**
Communication	*	***	*
Object	*	*****	***
State machine	***	****	**
Composite structure	*	*****	****
Component	*	***	*****
Deployment	**	**	*****
Package	***	**	****
Timing	*	***	***

described, even a technical problem can be described at the user level in MOPS. In any case, the problem space deals with all the work that takes place in understanding the problem in the context of the software system before any solution or development is attempted.

Typical activities that take place in MOPS include documenting and understanding the requirements, analyzing requirements, investigating the problem in detail, and perhaps optional prototyping and understanding the flow of the process within the business. Thus the problem space would focus entirely on what is happening with the business or the user. With the exception of prototyping in the problem space, where some code may be written for the prototype, no serious programming is expected when MOPS is created.

1.5.4.1 UML Diagrams in MOPS As a nontechnical description of what is happening with the user or the business, the problem space will need the UML diagrams that help the modeler understand the problem without going into technological detail. The UML diagrams that help express what is expected of the system, rather than how the system will be implemented, are of interest here. As shown in Table 1.2, these UML diagrams in the problem space are as follows:

Use case diagrams—provide the overall view and scope of functionality. The use cases within these diagrams contain the behavioral (or functional) description of the system.

Activity diagrams—provide a pictorial representation of the flow anywhere in MOPS. In MOPS, these diagrams work more or less like flowcharts, depicting the flow within the use cases or even showing the dependencies among various use cases.

Class diagrams—provide the structure of the domain model. In the problem space, these diagrams represent business domain entities (such as `Account` and `Customer` in a banking domain), not the details of their implementation in a programming language.

Sequence and state machine diagrams—occasionally used to help us understand the dynamicity and behavior of the problem better.

Interaction overview diagrams—recently added in UML 2.0, these provide an overview of the flow and/or dependencies between other diagrams.

Package diagrams—can be used in the problem space to organize and scope the requirements. Domain experts, who have a fairly good understanding not only of the current problem but also of the overall domain in which the problem exists, help provide a good understanding of the likely packages in the system.

1.5.5 Model of Solution Space (MOSS)

The solution space is primarily involved in the description of how the solution will be implemented. Figure 1.1 shows MOSS as a model that helps us understand and model the software solution that needs to be provided in response to MOPS. This solution model requires extra knowledge and information about the facilities provided by the programming languages, corresponding databases, middleware, Web application solutions and a number of other technical areas. Thus, MOSS contains a solution-level design expressed by technical or lower-level class diagrams, technical sequence diagrams, detailed state machine diagrams representing events and transitions, designs of individual classes and corresponding advance class diagrams. Object diagrams, communication diagrams and timing diagrams (the recent UML 2.0 addition) can also be occasionally used in MOSS.

1.5.5.1 UML Diagrams in MOSS Because MOSS is a technical description of how to solve the problem, the UML diagrams within MOSS are also technical in nature. Furthermore, even the diagrams drawn in MOPS are embellished in the solution space with additional technical details based on the programming languages and databases. As shown in Table 1.2, the primary diagrams used in MOSS are the class diagrams together with their lowermost details, including attributes, types of attributes, their initial values, signatures of the class operations (including their parameters and return values) and so on. These can be followed by diagrams like the sequence diagrams together with their detailed signatures, message types, return protocols and so on. Modelers may also use the communication diagrams for the same purpose as sequence diagrams. Occasionally state machine diagrams can be used to provide the dynamic aspect of the life cycle of a complex or very important object. Recently introduced timing diagrams show state changes to multiple objects at the same time, and composite structure diagrams depict the run-time structure of components and objects.

1.5.6 Model of Background Space (MOBS)

MOBS incorporates two major aspects of software development that are not covered by MOPS or MOSS: management and architecture. As shown in Figure 1.1, MOBS is an architectural model in the background space that influences models in both problem and solution spaces through constraints.

Of the two major aspects of work in the background space, management work relates primarily to planning. Planning deals mainly with the entire project and does not necessarily form part of the problem or solution space. In other words, management work in the background space includes issues from both problem and solution spaces but is not part of either of them. Several aspects of planning are handled in the background by the project manager. These include planning the project; resourcing the project's hardware, software, staff and other facilities; budgeting and performing cost-benefit analysis; tracking the project as it progresses through various iterations; and providing checkpoints for various quality-related activities. These background space activities are related to management work and are briefly discussed, along with other process aspects of quality, in Chapter 7. It is worth repeating here, though, that UML is not a management modeling language, and therefore does not provide direct notations and diagrams to document project plans and resources. The project planning aspect is best to deal with the process techniques as well as process tools (most containing Program, Evaluation & Review Technique [PERT] and Gantt charts and project-based task lists).

Architectural work, on the other hand, deals with a large amount of technical background work. This work includes consideration of patterns, reuse, platforms, Web application servers, middleware applications, operational requirements and so on. This background space also includes issues such as reuse of programs and system designs, as well as system and enterprise architecture. Therefore, work in this space requires knowledge of the development as well as the operational environment of the organization, availability of reusable architecture and designs, and how they might fit together in MOPS and MOSS.

1.5.6.1 UML Diagrams in MOBS The UML diagrams of interest in the background space are the ones that help us create a good system architecture that strives to achieve all the good things of object orientation. For example, reusability, patterns and middleware need to be expressed correctly in MOBS, and UML provides the means to do so. The importance of each of the UML diagrams in the background space is shown in Table 1.2. One should expect a large amount of strategic technical work in the background space that will consider the architecture of the current solution, the existing architecture of the organization's technical environment, the operational requirements of the system (i.e., the technical requirements of the system when it goes into operation, such as disk spaces, memory needs, CPU speeds), the needs of the system in terms of its stress, volume and bandwidth, and so on.

In order to complete the solution, it is necessary to relate the solution-level classes closely to the component diagrams drawn in the background space. These component diagrams contain the .EXEs and .DLLs and are closely associated with the solution-level class diagrams, providing the final steps in the system development exercise before the user runs the application. When run-time components are modeled, they result in composite structure diagrams, which may also be used in the background space to specify and discuss the architecture of a component or a class.

Increasingly, operational issues are being expressed properly using UML diagrams in the background space. UML provides help and support in modeling the operational environment (or deployment environment) of the system by means of deployment diagrams. Furthermore, a combination of component and deployment diagrams can provide a basis for discussions between the architects and designers of the system concerning where and how the components will reside and execute. Using the extension mechanisms of UML, one can develop diagrams that help express Web application architectures, including whether they should be thin-client or thick-client, the level of security needed on each node and the distributed aspects of the architecture, to name only a few issues. These background UML diagrams also have a positive effect on architecting quality (i.e., mapping quality to a good architecture) by providing standard means of mapping designs to existing and proven architectures. For example, an architectural pattern describing thin-client architecture is a much better starting point in terms of quality than designing such a solution from scratch.

1.6 VERIFICATION AND VALIDATION

Perry (1991) considers goals, methods and performance to be the three major aspects of quality. These strategic aspects of quality translate operationally into V&V techniques. Verification is concerned with the syntactic correctness of the software and models, whereas validation deals with semantic meanings and their value to the users of the system. V&V are quality techniques that are meant to prevent as well as detect errors, inconsistencies and incompleteness. V&V comprises a set of activities and checks that ensure that the model is correct. Based on Perry's definitions, verification focuses on ascertaining that the software functions correctly, whereas validation ensures that it meets the user's needs. Thus, verification comprises a separate set of activities that ensure that the model is correct. Validation works to ensure that it is also meaningful to the users of the system. Therefore, validation of models deals with tracing the software to the requirements.

Because of the subjective nature of quality, it cannot be easily quantified. However, one simple way to grapple with this subjectivity is to utilize a checklist-based approach as a first step in V&V of the quality aspects of a model. The correctness of the software is verified by a suite of checklists that deal with the syntax of the models, whereas the meaning and consistency of the software models are validated by creating a suite of checklists dealing with semantic checks. Thus, verification requires concrete skills like knowledge of the syntax; validation starts moving toward the abstract, as shown in Figure 1.2. Once augmented with aesthetic checks, this complete suite of checklists provides a quantifiable way of measuring quality, and it can be used as a benchmark for further developing qualitative understanding.

Having discussed the various quality aspects of modeling that are enhanced by UML, we now consider the manner in which these qualities of UML-based

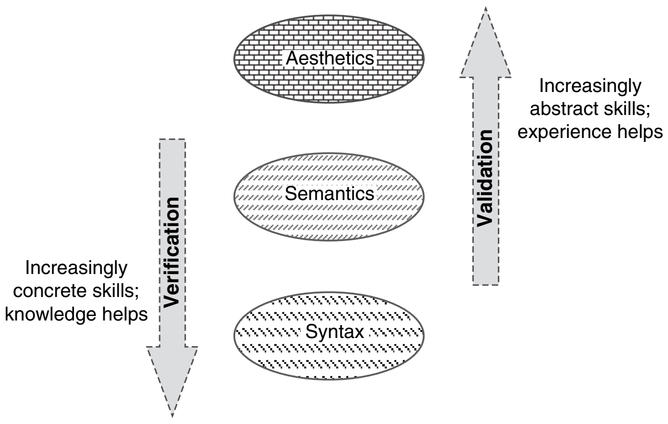
Aesthetics
Increasingly abstract skills; experience helps
Validation
Semantics
Verification
Increasingly concrete skills; knowledge helps
Syntax

Figure 1.2 Quality and V&V of models and their mapping to syntax, semantics and aesthetics.

models can be verified and validated. Some parts of V&V deal with the visual aspects of the model, others with its specification, construction and documentation.

Since UML is a language for visualization, it is appropriate to consider how the quality checks can be applied to UML-based diagrams and models. Therefore, the major part of V&V deals with the visual aspects of the model. This can lead not only to detection of errors in the model (quality checks that ensure validation of the model) but also appropriate quality assurance and process-related activities aimed at the prevention of errors. While recognizing the wide variety of definitions of quality in the software literature, we now start moving toward the basis for creating three types of V&V checks. For V&V of a software artifact, there are three levels of checks: syntax, semantics and aesthetics. These checks have close parallels to the quality approach of Lindland et al. (1994), who created a framework with three axes for quality assessment: language, domain and pragmatics. They translated these axes into syntactic quality, semantic quality and pragmatic quality, providing the theoretical background on which the current quality checks are built. While the syntax and semantic checks outlined here have close parallels to the work of Lindland et al., the aesthetic checks are also discussed by Ambler (2003) under the heading of "styles."

Building further on the framework of Lindland et al. (1994), the understanding of good-quality software modeling results in V&V of software models as follows:

- All quality models should be syntactically correct, thereby adhering to the rules of the modeling language (in our case, UML 2.0) they are meant to follow.
- All quality models should represent their intended semantic meanings and should do so consistently.
- All quality models should have good aesthetics, demonstrating the creativity and farsightedness of their modelers. This means that software models should be symmetric, complete and pleasing in what they represent.

The words "syntax," "semantics" and "aesthetics" are chosen to reflect the techniques or means of accomplishing the V&V of the models. One reason that these words correctly represent our quality assurance effort is that they relate directly to the UML models—especially those models that are created and stored in CASE tools. As a result, their quality can be greatly enhanced by applying the syntax, semantics and aesthetic checks to them. We will now consider these three categories of checks in more detail.

1.6.1 Quality Models—Syntax

All languages have a syntax. Latin and Sanskrit have their own syntax, and so do Java, XML and UML. However, two major characteristics of UML differentiate it from the other languages:

- UML is a visual language, which means that it has a substantial amount of notation and many diagram specifications.

- UML is a modeling language, which means that it is not intended primarily to be compiled and used in production of code (as programming languages are)—although the trend toward support for both "action semantics" in UML 2.0 and in MDA, both from the OMG, will likely involve the use of UML in this context in the future.

Needless to say, incorrect syntax affects the quality of visualization and specification, also, although a diagram itself cannot be compiled, incorrect syntax at the diagram level percolates down to the construction level, causing errors in creating the software code.

CASE tools are helpful to ensure that syntax errors are kept to a minimum. For example, on a UML class diagram, the rules of the association relationship, creation of default visibilities (e.g., private for attributes) and setting of multiplicities are examples of how CASE tools help to reduce syntax errors.

In UML-based models, when we apply syntax checks, we ensure that each of the diagrams that make up the model has been created in conformance with the standards and guidelines specified by OMG. We also ensure that the notations used, the diagram extensions annotated and the corresponding explanations on the diagrams all follow the syntax standard of the modeling language.

Figure 1.3 shows a simple example of a rectangle representing a dog. This rectangle is the notation for a class in UML. The syntax check on this diagram ensures that it is indeed a rectangle that is meant to represent animals (or other such things) in this modeling mechanism. The rectangle is checked for correctness, and we ensure that it is not an ellipse or for an arrowhead (both of which would be syntactically incorrect when using UML's notation) that is intended to represent the animal in question. In terms of UML models, a syntax check is a list of everything

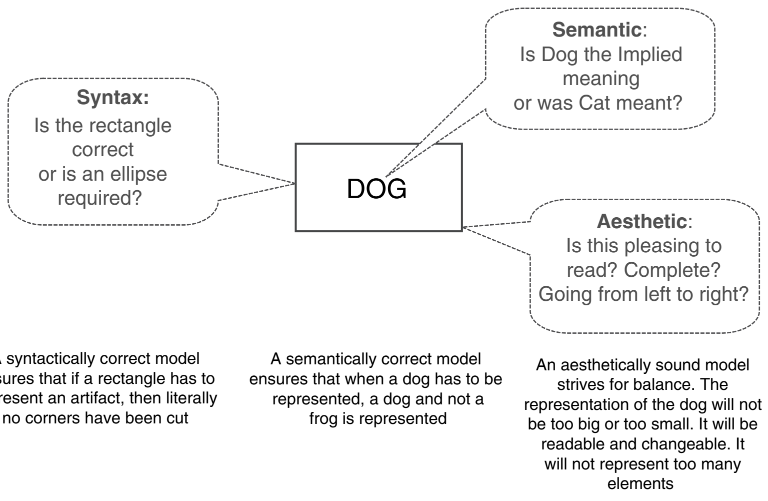
Semantic: Is Dog the Implied meaning or was Cat meant?
Syntax: Is the rectangle correct or is an ellipse required?
DOG
Aesthetic: Is this pleasing to read? Complete? Going from left to right?
A syntactically correct model ensures that if a rectangle has to represent an artifact, then literally no corners have been cut
A semantically correct model ensures that when a dog has to be represented, a dog and not a frog is represented
An aesthetically sound model strives for balance. The representation of the dog will not be too big or too small. It will be readable and changeable. It will not represent too many elements

Figure 1.3 Application of syntax, semantics and aesthetics.

that needs to be accomplished to achieve the syntax for the diagrams and associated artifacts of UML as laid out by OMG.

Permissible variations on these diagrams in complying with the meta-model can become a project-specific part of the syntax checks. Syntactic correctness greatly enhances the readability of diagrams, especially when these diagrams have to be read by different groups in different organizations in several countries (a typical software outsourcing scenario).

1.6.2 Quality Models—Semantics

While one of the qualities enhanced by rigorous syntax checks is the quality of construction (read "compilation"), one cannot be satisfied merely by a program that compiles and executes correctly yet does not consider the manner in which it is interpreted and understood. Such a model, although syntactically correct, would fail to achieve the all-important semantic correctness.

Consider, for example, Figure 1.3. Here, we expect to see a dog represented by a rectangle, with the word "dog" written in it. Writing the word "dog" within a rectangle might be syntactically correct, but it would be semantically wrong if the class `Dog` is actually representing an object cat (as it is in this example). If the class `Dog` is specified for an object cat, the meaning of the model is destroyed, however syntactically correct the model may be.

The semantic aspect of model quality ensures not only that the diagrams produced are correct, but also that they faithfully represent the underlying reality represented in the domain, as defined by Warmer and Kleppe (1998). In UML, for example, the business objectives stated by the users should be correctly reflected in the use case diagrams, business rules, constraints, and pre- and postconditions documented in the corresponding use case documentation.

Once again, models in general are not executable; therefore, it is not possible to verify and validate their purpose by simply "executing" them, as one would the final software product (the executable). Consequently, we need to identify alternative evaluation techniques. In this context, the traditional and well-known quality techniques of walkthroughs and inspections (e.g., Warmer and Kleppe, 1998; Unhelkar, 2003) are extremely valuable and are used more frequently and more rigorously than for syntax checking.

Another example of such techniques, for instance as applied to use case models in UML, is that we anthropomorphize[2] each of the actors and use cases and act through an entire diagram as if we were the objects themselves. We can insist that testers walk through the use cases, verify the purpose of every actor and all use cases, and determine whether they depict what the business really wants. This is the semantic aspect of verifying the quality of a UML model, supplemented, of course, by the actual (non-UML) use case descriptions themselves (e.g., Cockburn, 2001).

[2]Personify—by assuming that the actors are alive and conducting a walkthrough/review with business users, developers and testers.

1.6.3 Quality Models—Aesthetics

Once the syntax and the semantics are correct, we need to consider the aesthetics of the model (e.g., Ambler, 2003). Very simply, aesthetics implies style. Often, while reading a piece of code, one is able to point out the style or programming and hence trace it to a specific programmer or a programming team. Although the code (or, for that matter, any other deliverable) may be accurate (syntactically) and meaningful (semantically), difference still arises due to its style. The style of modeling has a bearing on the models' readability, comprehensibility and so on. One example of a factor that affects style is granularity (discussed in detail in Chapter 6). In good OO designs, the level of granularity needs to be considered, as it strongly affects understandability (Miller, 1956). For example, in Figure 1.3, how many rectangles (classes) are there on a diagram (as against the previous two checks: "Is that a class notation?" and "What is the meaning behind this class?")? It is, of course, possible that a system with 10 class diagrams, each with 10 classes and numerous relationships, may accurately represent a business domain model—although such large numbers should be regarded as a warning (e.g., Henderson-Sellers, 1996). In another example, one class diagram may have 20 classes (not wrong from a UML viewpoint, but ugly) and another class diagram may have only 1, albeit an important and large one. This aesthetic size consideration is studied in terms of the granularity of the UML models, as described by Unhelkar and Henderson-Sellers (1995), and requires a good metrics program within the organization to enable it to improve the aesthetics of the model. Such a model will then offer a high level of customer satisfaction, primarily to the members of the design team but also in their discussions with the business enduser(s).

1.6.4 Quality Techniques and V&V Checks

The three aspects of quality checks—syntax, semantics and aesthetics—should not be treated as totally independent of each other. A change in syntax may change the meaning or semantics of a sentence or diagram. While syntax is checked minutely for each artifact, an error in syntax may not be limited to the error in the language of expression.

This also happens in UML, where syntax and semantics may depend on each other. For example, the direction of an arrow showing the relationship between two classes will certainly affect the way that class diagram is interpreted by the end user. Similarly, aesthetics or symmetry of diagrams facilitates easier understanding (e.g., Hay, 1996), making the semantics clearer and the diagrams more comprehensible to their readers.

This brings us to the need to consider the various traditional quality techniques of walkthroughs, inspections, reviews and audits in the context of the V&V checks of syntax, semantics and aesthetics, as shown in Figure 1.4.

Walkthroughs—may be performed individually, and help weed out syntax errors (more than semantic errors).

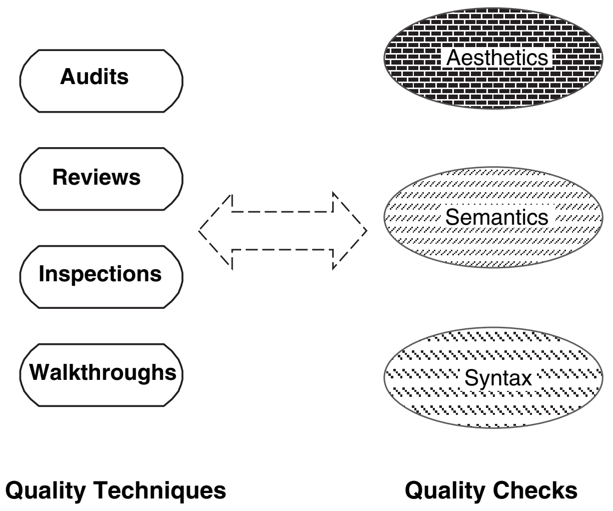

Figure 1.4 V&V checks and quality techniques.

Inspections—are more rigorous than walkthroughs, are usually carried out by another person or party, and can identify both syntax and semantic errors.

Reviews—increase in formality and focus on working in a group to identify errors. The syntax checks are less important during reviews, but the semantics and aesthetics start becoming important.

Audits—formal and possibly external to the project and even the organization. As a result, audits are not very helpful at the syntax level, but they are extremely valuable in carrying out aesthetic checks of the entire model.

1.7 QUALITY CHECKS AND SKILLS LEVELS

As shown in Figure 1.2, while the syntax can be verified by anyone who has sufficient knowledge of UML, the semantics of each of the UML diagrams and the models that these diagrams describe need a little more experience. It is therefore important to include the users in all quality checks and to encourage them to participate in all quality walkthroughs, inspections and playacting (anthropomorphizing) in verifying the semantics of each of the models. The UML experience here fully supports the participatory role of the user envisaged by Warmer and Kleppe (1998).

The aesthetic aspect of model quality requires a combination of knowledge and experience. In ensuring the aesthetics of the UML models created, we require knowledge not only of UML and of the business, but also of the CASE tools and the environment in which they have been used. We need experience with more than one project before we can successfully apply the aesthetic aspect of model quality to the UML model.

1.8 LEVELS OF QUALITY CHECKS TO UML DIAGRAMS

Levels of checks mean that while syntax, semantics and aesthetic checks are applied to the UML diagrams (Figure 1.5), these checks are also applied in various ways to the entire model, which is made up of many diagrams. Alternatively, they can be applied to a single artifact within a diagram. Thus, it is not necessary to have all types of checks that apply to all artifacts, diagrams and models produced.

However, this understanding of the levels of checks is helpful in focusing on the intensity of the checks and in ensuring that quality improvement efforts are well balanced. This is explained further in the following subsections.

The modeling constructs offered by UML and the corresponding quality checks at the three levels are as follows:

a. The individual elements, or "things" that make up the diagrams. The artifacts (or things) and the specifications of these artifacts should have the syntax, semantics and aesthetics checks applied as far as possible. This comprises "ground-level" or highly detailed checking. In these checks, the syntax of the artifacts is checked most intensely.
b. The UML diagrams and the validity of their syntax, semantics and aesthetics. This is the equivalent of a "standing view" of the model being verified and validated, with an intensive check of the semantics of diagrams.
c. A combination of interdependent diagrams called a "model". The V&V of the entire model, made up of the relevant UML diagrams, their specifications, and so on, includes syntax, semantics and aesthetics checks. This is the "bird's-eye view," allowing checks of symmetry and consistency, resulting in aesthetic quality checks being applied most intensely.

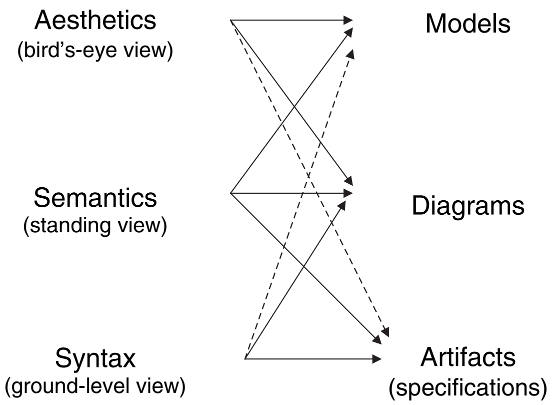

Figure 1.5 Syntax, semantics and aesthetics checks verify and validate UML artifacts, diagrams and models.

It is not necessary to apply all types of checks to all of the artifacts, diagrams and models produced.

1.8.1 Syntax Checks and UML Elements (Focus on Correctness)

When we say that we want to apply syntax checks to a use case diagram, what exactly do we mean? Are we checking the use case itself and its specification, or are we checking whether the "extends" relationship arrow in a use case diagram is pointing correctly to the use case being extended? This question leads us to expand our V&V effort to levels beyond just one diagram.

In syntax checks, we are looking at the ground-level view of the models. This includes the artifacts and elements of UML, as well as their specifications and documentation. Furthermore, when we check the syntax of these elements, we focus primarily on the correctness of representation as mandated by UML. Therefore, during syntax checks the semantics, or the meaning behind the notations and diagrams, are not the focus of checking.

For example, consider a class diagram that contains `Car` as a class. The syntax check of the correctness of this artifact would be something like this:

- Is the `Car` class represented correctly by attributes and operations?
- Do the attributes have correct types and do the operations have correct signatures?
- Is the `Car` class properly divided into three compartments?
- Is the `Car` class compilable? (This syntax check will apply in the solution space.)

In UML terms, when we start applying syntax checks to a use case diagram, we first apply them to the artifacts or elements that make up the diagram, such as the actors and the use cases. In a class diagram, these basic syntax checks apply to a class first and whatever is represented within the class. Since these artifacts are the basic building blocks from which the diagrams and models are created in UML, checking them in terms of correctness of the UML syntax is the first thing that should be done in any quality control effort.

This syntax check for an element or artifact is followed by a check of the validity of the diagram itself. Here we do not worry about whether, say, the specifications of the use case itself follow a standard and whether the use case semantically represents what it is meant to represent. Instead of focusing on one element at this level, we inspect the entire diagram and ensure that it is syntactically correct.

If these syntax checks for the elements and the diagrams that comprise them are conducted correctly, they ensure the correctness of the UML diagrams. As a result, the intensity of syntax checks will be reduced when the entire model is checked.

1.8.2 Semantic Checks and UML Diagrams (Focus on Completeness and Consistency)

Semantic checks deal with the meaning behind an element or a diagram. Therefore, this check focuses not on the correctness of representation but on the completeness of the meaning behind the notation. In the example of the `Car` class considered above, the semantic check for the model of `Car` would be: "Does the `Car` class as named in this model actually represent a car or does it represent a garbage bin?" It is worth noting here that should a collection of garbage bins be named as a `Car` class, so long as it has a name, an attribute and operation clearly defined, the UML syntax checks for the `Car` class will be successful. It is only at the semantic level that we can figure out that something is wrong because in real life the name `Car` does not represent a collection of garbage bins.

Because the meaning of one element of UML depends on many other elements and on the context in which it is used, therefore, semantic checks are best performed from a standing-level view of the UML models. This means that we move away from the ground-level check of the correctness of representation and focus on the purpose of representation. Needless to say, when we stand up from the ground (where we inspect the syntax), a lot more become visible. Therefore, it is not just one element on the diagram but rather the entire diagram that becomes visible and important. Semantic checks, therefore, become more intense at the diagram level rather than just at an element level.

Taking the `Car` example further, semantic checks also deal with consistency between diagrams, which includes, for example, dependencies between `doors` and `engine` and between `wheel` and `steering`. In UML terms, while a class `door` may have been correctly represented (syntactically correct) and may mean a `door` (semantically correct), the dependencies between `door` and `car`, or between `door` and `driver` (or even between `door` and `burglar`), will need a detailed diagram-level semantic check. This check will also include many cross-diagram dependency checks that extend the semantic check to more than one diagram. Semantic checks also focus on whether this class is given a unique and coherent set of attributes and responsibilities to handle or whether it is made to handle more responsibilities than just `Car`. For example, do the `Driver`-related operations also appear in `Car`? This would be semantically incorrect. Thus, semantic checks apply to each of the UML diagrams intensely, as well as to the entire model.

1.8.3 Aesthetic Checks and UML Models (Focus on Symmetry and Consistency)

As noted in the preceding two subsections, the correctness and completeness of UML elements and the corresponding individual diagrams are ensured by applying detailed syntax and semantic checks to them. The aesthetic checks of these diagrams and models add a different dimension to the quality assurance activities, as they deal not with correctness or completeness but rather with the overall consistency and symmetry of the UML diagrams and models. They are best done with a birds-eye view of the model. Because these checks occur at a very high level, far more is visible—not just one diagram, but many diagrams, their interrelationships, and their look and feel. This requires these aesthetic checks to be conducted at certain "checkpoints," where a certain amount of modeling is complete. Therefore, aesthetic checks also require some knowledge and understanding of the process being followed in the creation of the models and the software (briefly discussed in Chapter 7). The process ensures that the aesthetic checks are applied to the entire model rather than to one element or diagram.

In UML terms, the aesthetic checks of the `Car` class involve checking the dependency of `Car` on other classes and their relationships with persistent and graphical user interface (GUI) class cross-functional dependencies. This requires cross-checks between various UML diagrams that contain the `Car` class as well as checks of their consistency. Furthermore, aesthetic checks, occurring at a birds-eye level, focus on whether the `Car` class has too many or too few attributes and responsibilities. For example, if the `Car` class has too many operations, including that of "driving itself," the entire model would become ugly. Thus, a good understanding of the aesthetic checks results in diagrams and models that do not look ugly, irrespective of their correctness.

Finally, aesthetic checks look at the entire model (MOPS, MOSS, MOBS or any other) to determine whether or not it is symmetric and in balance. If a class diagram in a model has too many classes, aesthetic checks will ensure redistribution of classes.

Thus we see that, together, the syntax, semantic and aesthetic checks ensure that the artifacts we produce in UML, the diagrams that represent what should be happening in the system, and the models that contain diagrams and their detailed corresponding documentation are all correct, complete and consistent.

1.9 MODEL-DRIVEN ARCHITECTURE (MDA) AND QUALITY

MDA (OMG, 2004) is the latest initiative by the OMG to create an application architecture that is reusable in developing applications. The purpose of MDA is to provide the basic infrastructure and leave developers free to concentrate on solving application problems. MDA enables developers to look at the challenges of requirements modeling, development, testing and portability of deployed systems. MDA provides the basis for this effort and, at the same time, helps to increase the reusability of architecture by separating specification of the system's operation from the details of the way that the system uses the capabilities of its platform. The MDA initiative depends strongly on UML. Some authors bring the initiative close to the discussion of executable UML (see Fowler, 2003). The key components of MDA are the computation independent model (CIM), platform independent model (PIM) and platform specific model (PSM). PIM and PSM are effectively in the problem and solution spaces, as shown in Figure 1.1.

1.10 PROTOTYPING AND MODELING SPACES

An introduction to the concepts of quality would not be complete without mentioning prototyping. A prototype is a type of model, and it is advisable to use it in conjunction with the UML models to achieve overall good quality in the project. Prototypes can be created in each of the three modeling spaces to validate and verify the requirements as well as extract complete and correct requirements. Here are some brief comments on the nature of prototypes in each of the three modeling spaces.

MOPS has its own prototype, which is called the "functional prototype." This contains the user interface prototype. An example of using the prototype in MOPS is the use of the class responsibility collaborator (CRC) technique in requirements modeling. Each of the cards representing the classes can be used in role-playing a use case and the domain-level classes and their responsibilities extracted. Another well-known example of a prototype in MOPS is a "dummy" executable of a system, using screens and displaying their look, feel and navigation. Functional prototypes can thus be used to set the expectations of both users and management. By showing what a system can and cannot do early in MOPS, it is possible to reiterate the objectives and the scope of the system.

The prototype in MOSS is that of the technology. This implies testing programming languages and databases. While the MOPS prototype need not be an executable, the one in MOSS probably would. For example, a technical prototype would have an example Java class created that would be made to run on a potential client machine. The prototype would also experiment with running a small application on a server by trying various languages, like Java and C++, to handle the server-side capabilities. Potential reuse through patterns and reusable components is also facilitated by the technical prototypes created in MOSS.

Prototypes in the background space would test architectural requirements such as bandwidth and operating systems. Some aspects of performance, volume, stress, security and scalability would also be managed between the prototypes, MOSS and MOBS.

The architectural prototype could be the same prototype created in the solution space to explore how well the overall architecture of the system fits in with the rest of the system's and the organization's environment. Unlike the prototype in the problem space, this prototype would usually be an executable piece of software that experiments with various components, databases, networks and security protocols, to name but a few.

DISCUSSION TOPICS

1. What is the importance of modeling in enhancing the quality of software projects? How is the importance of modeling different in enhancing the quality of the software models themselves?
2. What are the limitations of modeling? What rules should we adhere to when using modeling in software projects?
3. What are the various levels of quality, and how does model quality fit into these quality levels?

4. How does modeling help in projects where there is a large amount of existing legacy code (integration, creating models of existing applications)?
5. Describe the three suggested modeling spaces within software and how they relate to UML-based modeling.
6. What are the different aspects of quality enhanced by UML?
7. Discuss the important UML 2.0 diagrams in MOPS.
8. Discuss the important UML 2.0 diagrams in MOSS.
9. Discuss the important UML 2.0 diagrams in MOBS.
10. What is verification and validation? Discuss the difference between the two.
11. How does the aesthetic aspect of quality models differ from their syntax and semantic aspects?
12. What is an appropriate quality technique to verify syntax?
13. What is an appropriate quality technique to verify semantics?
14. What is an appropriate quality technique to verify aesthetics?
15. In addition to the knowledge of UML, what else is needed in ensuring the semantic quality of UML?
16. How is prototyping helpful in modeling? Discuss this with respect to the three modeling spaces and the models created there.
17. What is MDA? How does it relate to the three modeling spaces?

REFERENCES

Ambler, S. *UML Style Guide*. Cambridge: Cambridge University Press, 2003.

Booch, G., Rumbaugh, J., and Jacobson, I. *The Unified Modelling Language User Guide*. Reading, MA: Addison-Wesley, 1999.

Cockburn, A. *Writing Effective Use Cases*. Boston, MA: Addison-Wesley, 2001.

Fowler, M. *Patterns of Enterprise Application Architecture*. Reading, MA: Addison-Wesley Professional, 2003.

Glass, R. *Facts and Fallacies of Software Engineering*. Reading, MA: Addison-Wesley, 2003.

Hay, D.C. *Data Model Patterns: Conventions of Thoughts*. New York: Dorset House, 1996.

Henderson-Sellers, B. *Object Oriented Metrics: Measures of Complexity*. Upper Saddle River, NJ: Prentice Hall, 1996.

Jacobson, I., Christerson, M., Jonsson, P. and Övergaard, G. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: Addison-Wesley, 1992, pp. 524.

Miller, G. "The Magical Number Sever, Plus or Minus Two: Some Limits on our Capacity for Processing Information," *The Psychological Review*, 63(2), 1956, pp. 81–97.

Mellor, S.J., and Balcer M.J. *Executable UML: A Foundation for Model Driven Architecture*. Reading, MA: Addison-Wesley, 2002.

Mellor, S., Scott, K., Uhl, A., and Weise, D. *MDA Distilled: Principles of Model-Driven Architecture*. Reading, MA: Addison-Wesley, 2004.

Lindland, O.I., Sindre, G., and Sølvberg, A. "Understanding Quality in Conceptual Modeling," *IEEE Software* (March 1994), 42–49.

OMG. OMG Unified Modeling Language Specification, Version 1.4, September 2001. OMG document formal/01-09-68 through 80 (13 documents) [online]. Available at http://www.omg.org (2001).

OMG, Model Driven Architecture Initative; accessed 2004.

Perry, W. *Quality Assurance for Information Systems*. MA: QED Information Sciences, 1991.

Rosenberg, D., and Scott, K. *Use Case Driven Object Modelling with the UML.* Reading, MA: Addison-Wesley, 1999.

Unhelkar, B. *After the Y2K Fireworks*. Boca Raton, FL: CRC Press, 1999.

Unhelkar, B. *Process Quality Assurance for UML-Based Projects*. Boston: Addison-Wesley, 2003.

Unhelkar, B., and Henderson-Sellers, B. "ODBMS Considerations in the Granularity of Reuseable OO Design," *Proceedings of TOOLS15 Conference*, C. Mingins and B. Meyer, eds. Upper Saddle River, NJ: Prentice-Hall, 1995, pp. 229–234.

Unhelkar, B., and Henderson-Sellers, B. "Modelling Spaces and the UML," *Proceedings of the IRMA (Information Resource Management Association) Conference*, New Oreleans, 2004.

Warmer, J., and Kleppe, A. *The Object Constraint Language. Precise Modeling with UML.* Reading, MA: Addison-Wesley, 1998.